CSE 333
Section 7

Casting & Client-Side Networking

UNIVERSITY of WASHINGTON

Logistics

Tonight:

HW3 @ 11:59 pm

Monday:

Exercise 15 @ 10:30 am

Casting in C++

Four different casts that are more explicit:

1. `static_cast<to_type>(expression)`
2. `dynamic_cast<to_type>(expression)`
3. `const_cast<to_type>(expression)`
4. `reinterpret_cast<to_type>(expression)`

When programming in C++, you should use these casts!

Static Cast

```
static_cast<to_type>(expression)
```

Used to:

1) Convert pointers of *related* types

```
Base* b = static_cast<Base*>(new Derived);
```

- compiler error if types aren't related

2) Non-pointer conversion

```
int qt = static_cast<int>(3.14);
```

Static Cast

```
static_cast<to_type>(expression)
```

[!] Be careful when *casting up*:

```
Derived* d = static_cast<Derived*>(new Base);
d->y = 5;
```

- compiler will let you do this
- dangerous if you want to do things defined in **Derived**, but not in **Base**!

Dynamic Cast

```
dynamic_cast<to_type>(expression)
```

Used to:

1) Convert pointers of *related* types

```
Base* b = dynamic_cast<Base*>(new Derived);
```

- *compiler* error if types aren't related
- at *runtime*, returns **nullptr** if it is actually an unsafe upwards cast:

```
Derived* d = dynamic_cast<Derived*>(new Base);
```

Const Cast

```
const_cast<to_type>(expression)
```

Used to:

1) Add or remove const-ness

```
const int x = 5;
const int *ro_ptr = &x
int *ptr = const_cast<int*>(ro_ptr);
```

Reinterpret Cast

```
reinterpret_cast<to_type>(expression)
```

Used to:

1) Cast between *incompatible* types

```
int* ptr = 0xDEADBEEF;
int64_t x = reinterpret_cast<int64_t>(ptr);
```

- types must be of same size
- refuses to do float-integer conversions

Exercise 1

```cpp
class Base {
 public:
  int x;
};
```

```cpp
class Derived : public Base {
 public:
  int y;
};
```

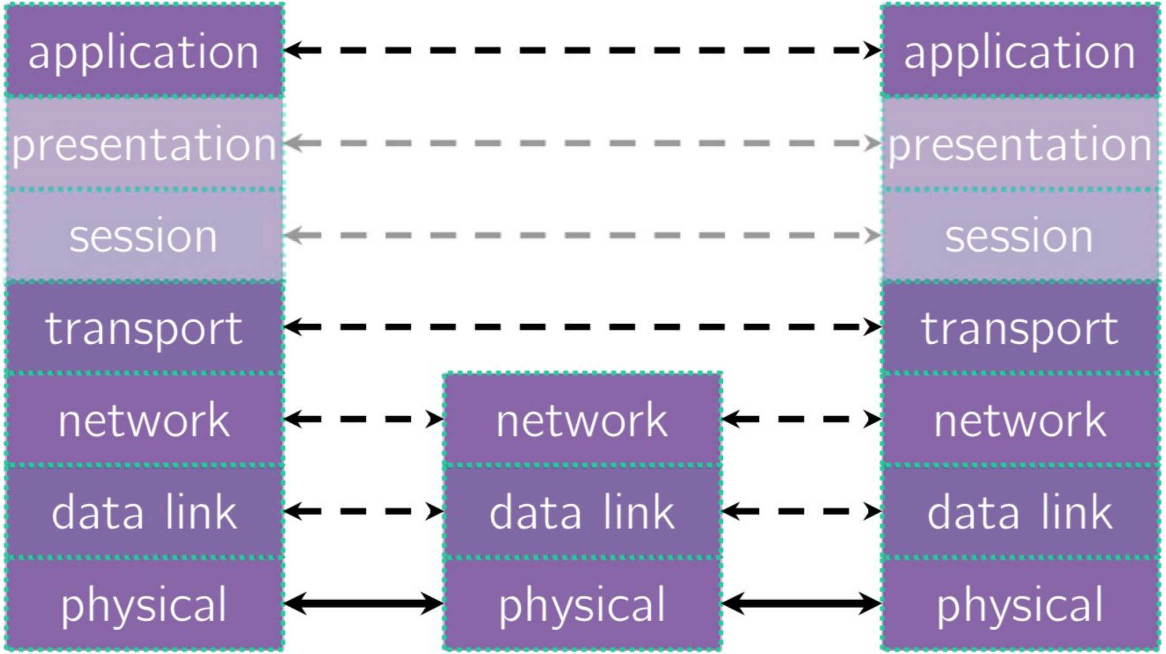
```cpp
int64_t x = 0x7fffffffe870;
char* str = ___reinterpret_cast<char *>___(x);
```

```cpp
void foo(Base *b) {
  Derived *d = ___dynamic_cast<Derived *>___(b);
  // additional code omitted
}
```

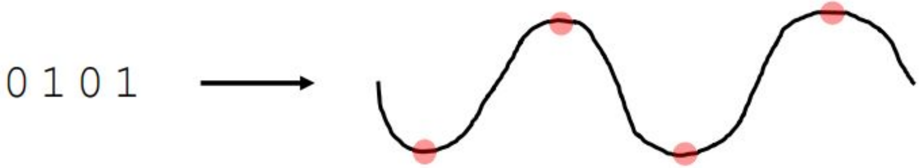
```cpp
Derived *d = new Derived;
Base *b = ___static_cast<Base *>___(d);
```

```cpp
double x = 64.382;
int64_t y = ___static_cast<int64_t>___(x);
```

Computer Networks: A 7-ish Layer Cake

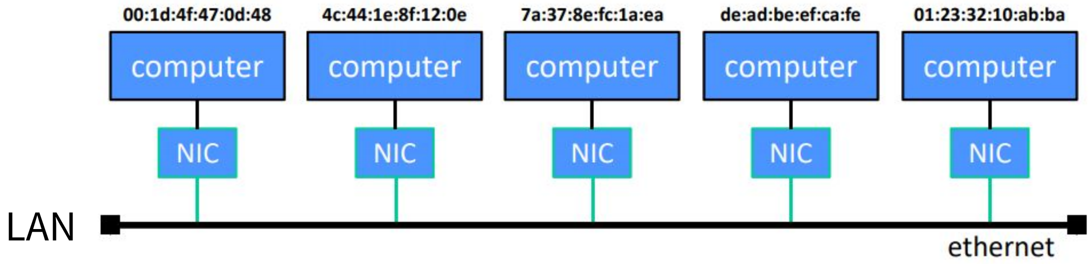


Computer Networks: A 7-ish Layer Cake



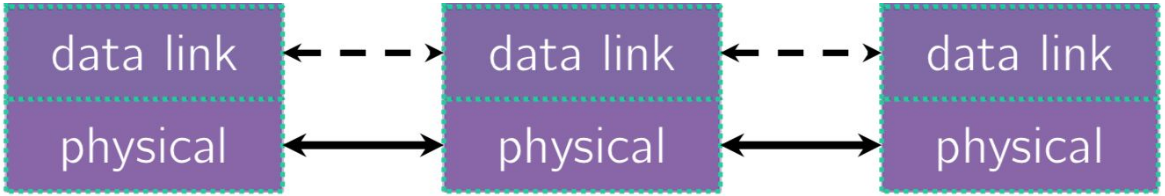
bit encoding at signal level

Computer Networks: A 7-ish Layer Cake

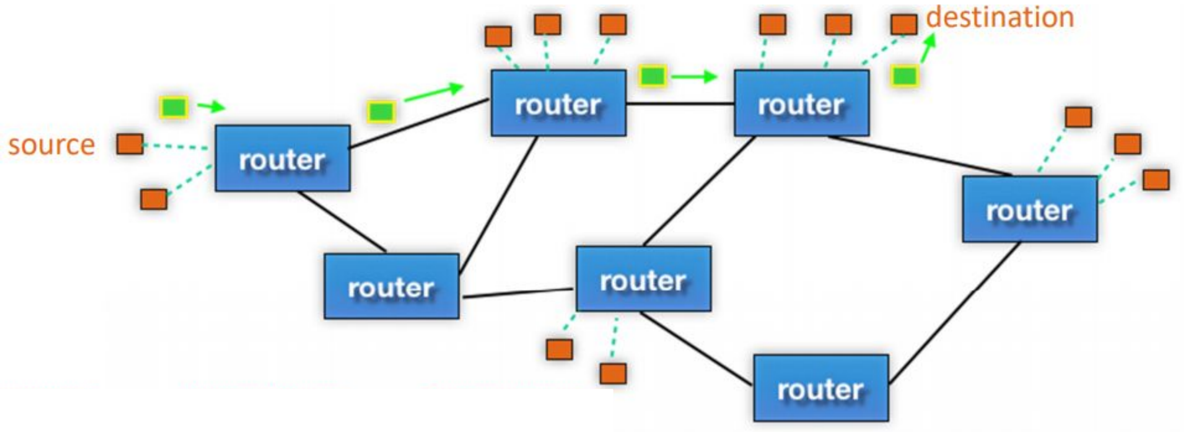


multiple computers on a local network

bit encoding at signal level



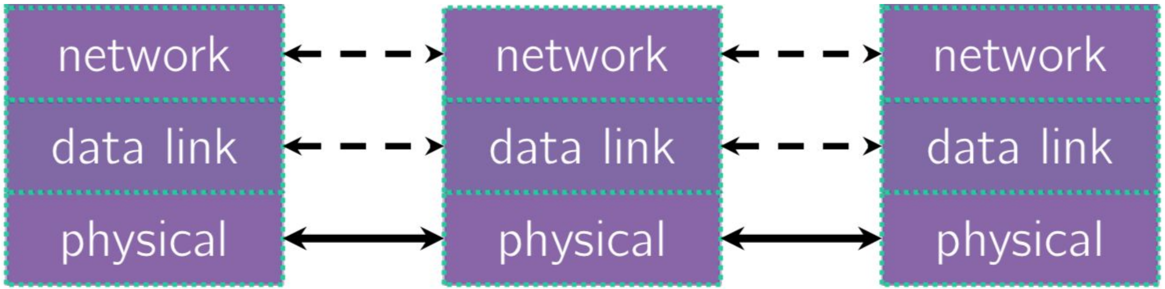
Computer Networks: A 7-ish Layer Cake



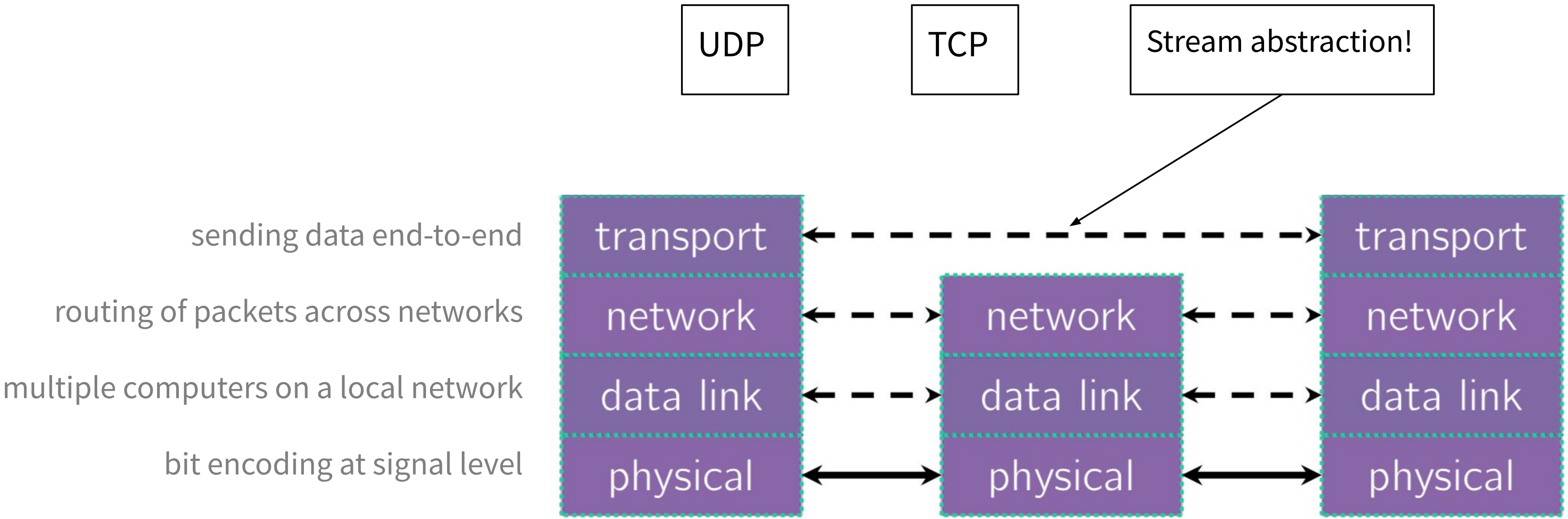
routing of packets across networks

multiple computers on a local network

bit encoding at signal level



Computer Networks: A 7-ish Layer Cake



Computer Networks: A 7-ish Layer Cake

HTTP

DNS

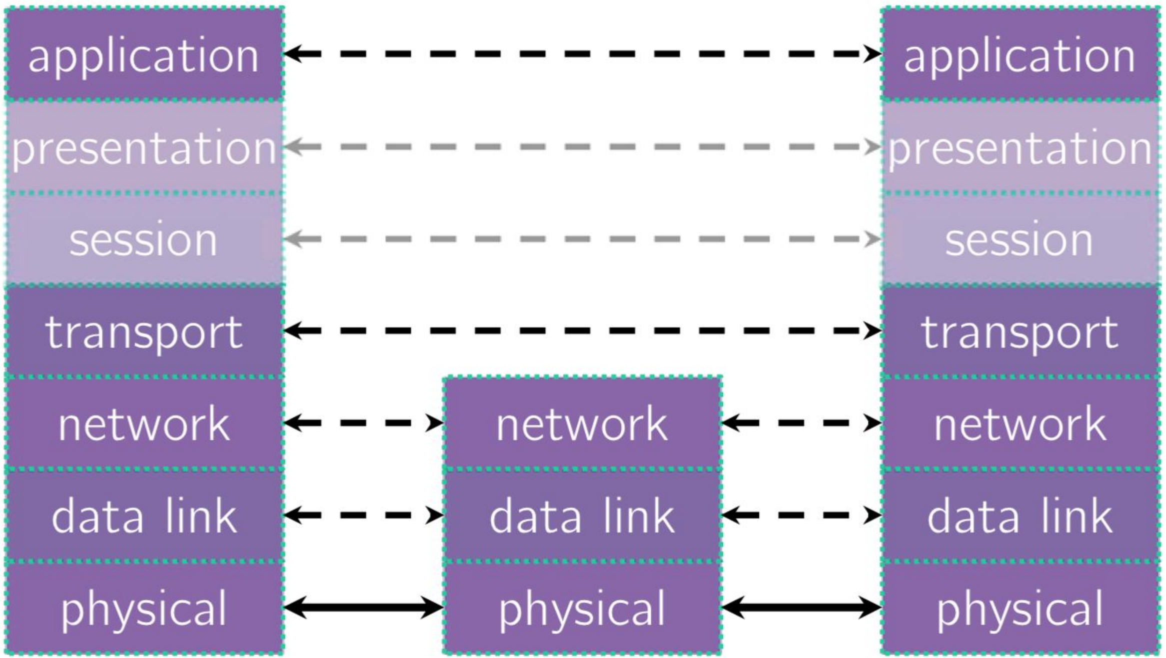
format/meaning of messages

sending data end-to-end

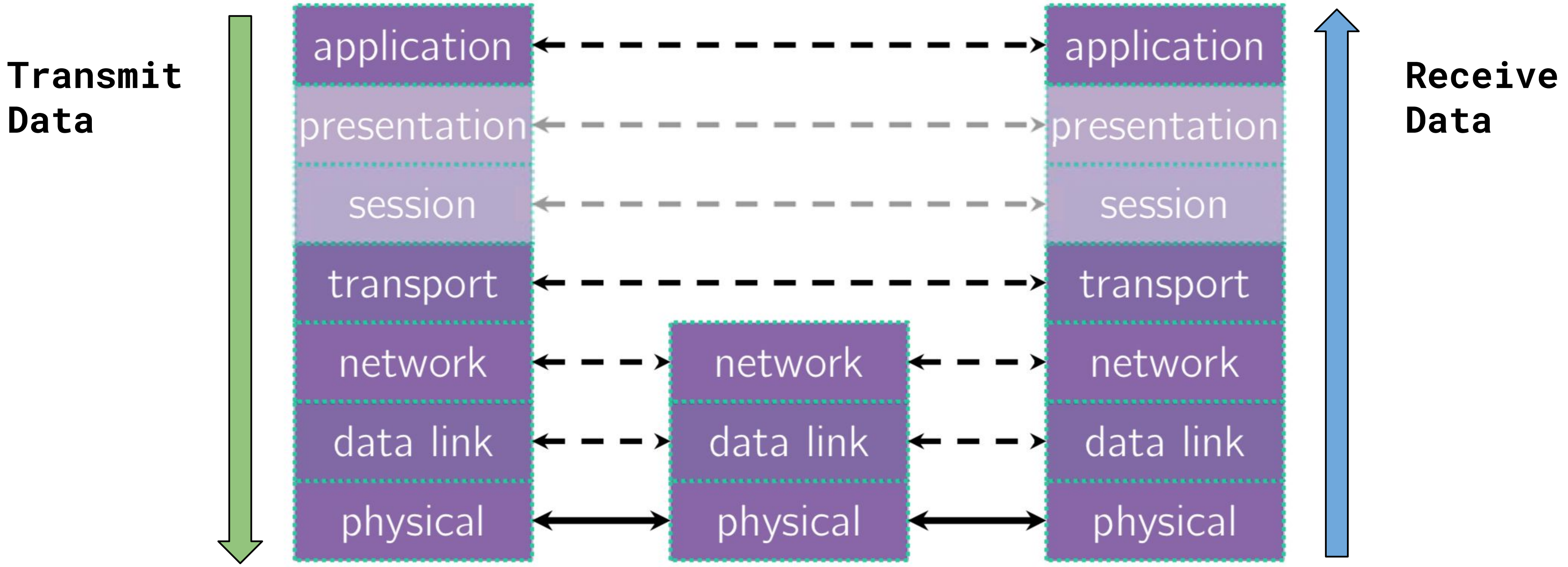
routing of packets across networks

multiple computers on a local network

bit encoding at signal level



Data flow


Transmit
Data
application
presentation
session
transport
network
data link
physical
network
data link
physical
application
presentation
session
transport
network
data link
physical
Receive
Data

Exercise 2

Exercise 2

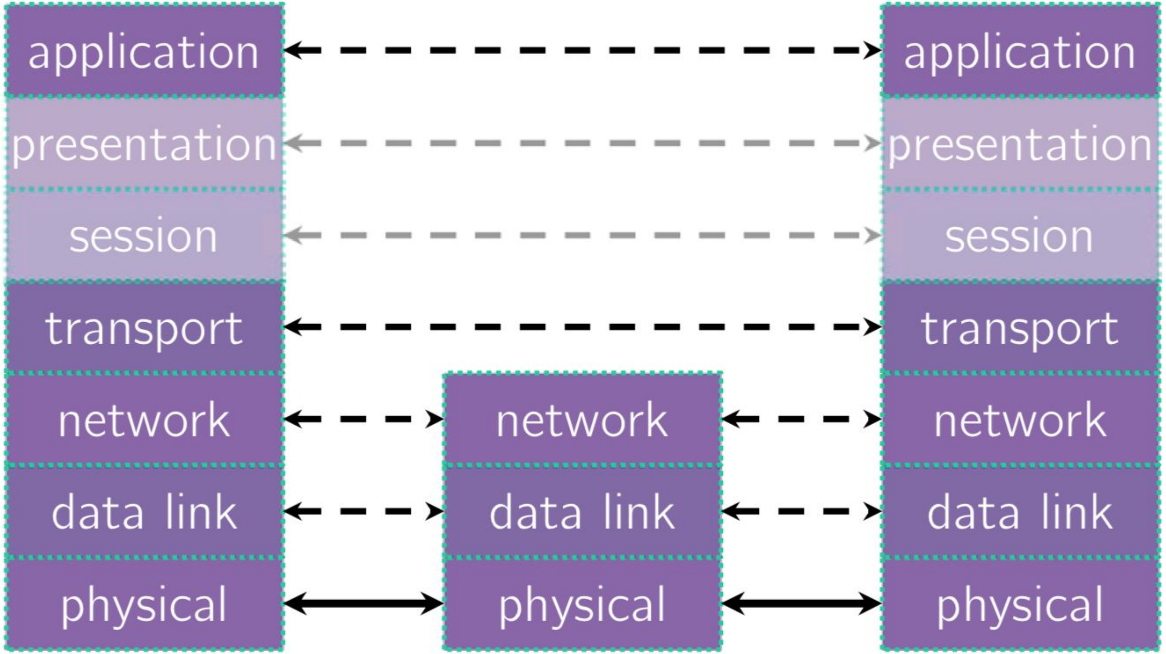
format/meaning of messages

sending data end-to-end

routing of packets across networks

multiple computers on a local network

bit encoding at signal level



Exercise 2

Vote in Zoom!

yes = application layer

no = transport layer

go slower = network layer

go faster = link layer

- DNS: Translating between IP addresses and host names. (Application Layer)
- IP: Routing packets across the Internet. (Network Layer)
- TCP: Reliable, stream-based networking on top of IP. (Transport Layer)
- UDP: Unreliable, packet-based networking on top of IP. (Transport Layer)
- HTTP: Sending websites and data over the Internet. (Application Layer)

TCP versus UDP

Transmission Control Protocol(TCP)

- Connection oriented Service
- Reliable and Ordered
- Flow control

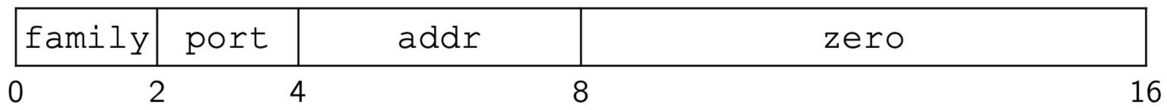
User Datagram Protocol(UDP)

- Connectionless service
- Unreliable packet delivery
- Faster
- No feedback

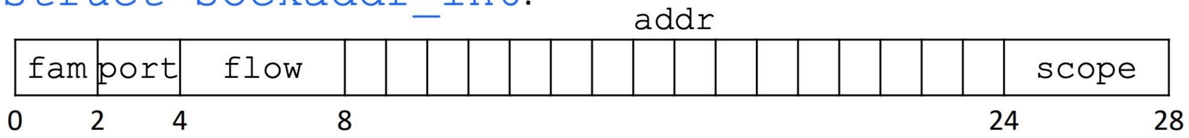
Sockets

- Just a file descriptor for network communication
- Types of Sockets
  - Stream sockets (TCP)
  - Datagram sockets (UDP)
- Each socket is associated with **a port number** and **an IP address**
  - Both port and address are stored in network byte order (big endian)

`struct sockaddr_in`:



`struct sockaddr_in6`:



Sockets

struct sockaddr (pointer to this struct is used as parameter type in system calls)

fam	????

....

struct sockaddr_in (IPv4)

fam	port	addr	zero

16

struct sockaddr_in6 (IPv6)

fam	port	flow	addr	scope

28

struct sockaddr_storage

fam	

Big enough to hold either

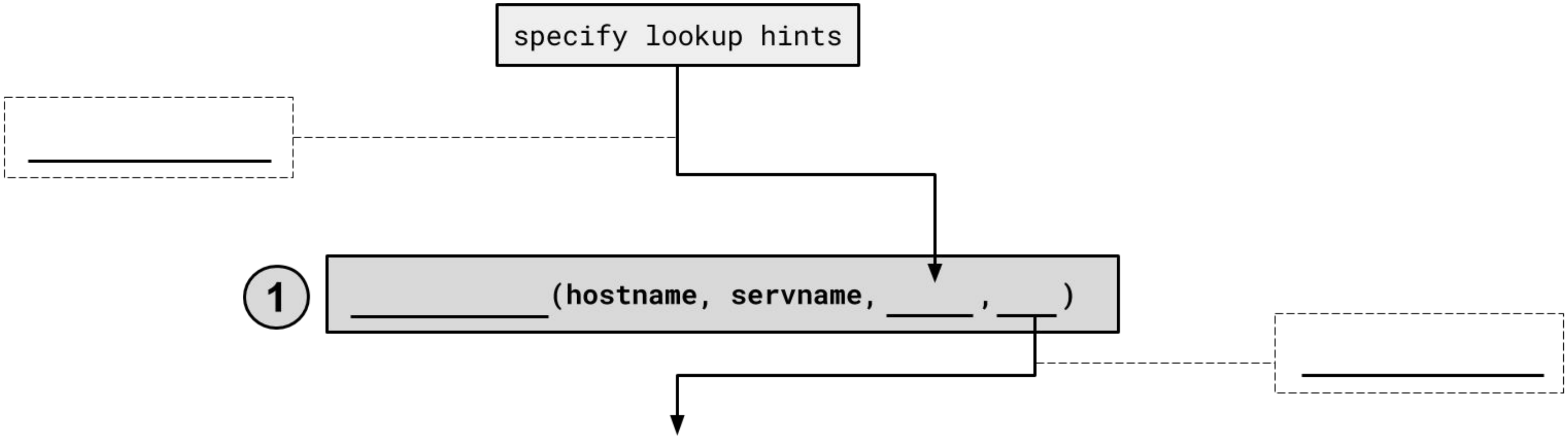
Byte Ordering and Endianness

- **N**etwork Byte Order (Big Endian)
    - The most significant byte is stored in the highest address
- **H**ost byte order
    - Might be big or little endian, depending on the hardware
- To convert between orderings, we can use
    - `uint16_t htons (uint16_t hostlong);`
    - `uint16_t ntohs (uint16_t hostlong);`

    - `uint32_t htonl (uint32_t hostlong);`
    - `uint32_t ntohl (uint32_t hostlong);`

Exercise 3

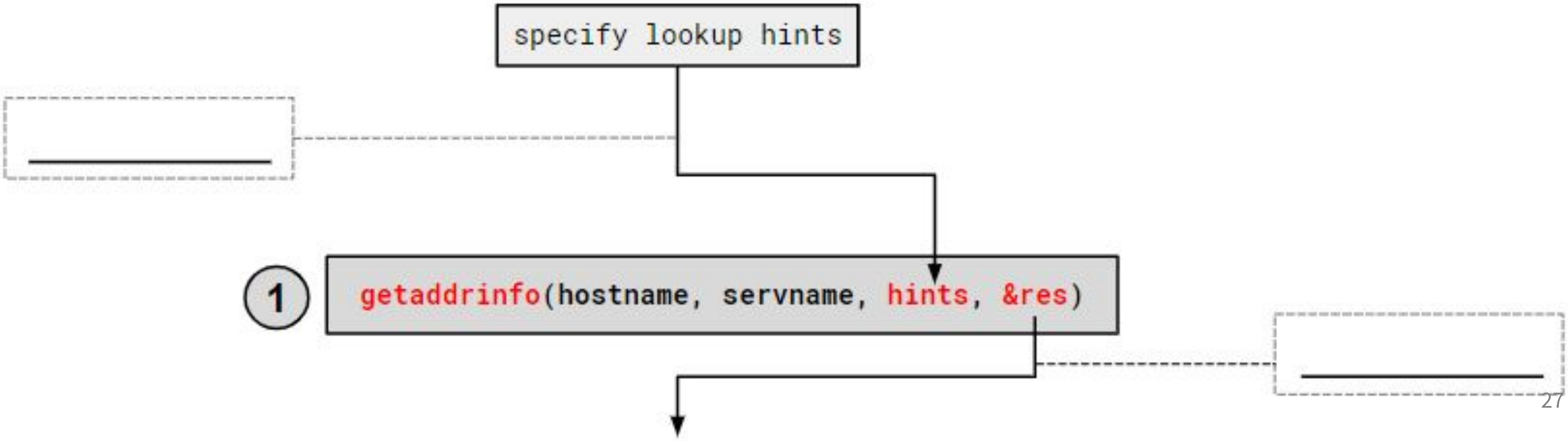
1.



1. getaddrinfo()

```
int getaddrinfo(const char *hostname,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

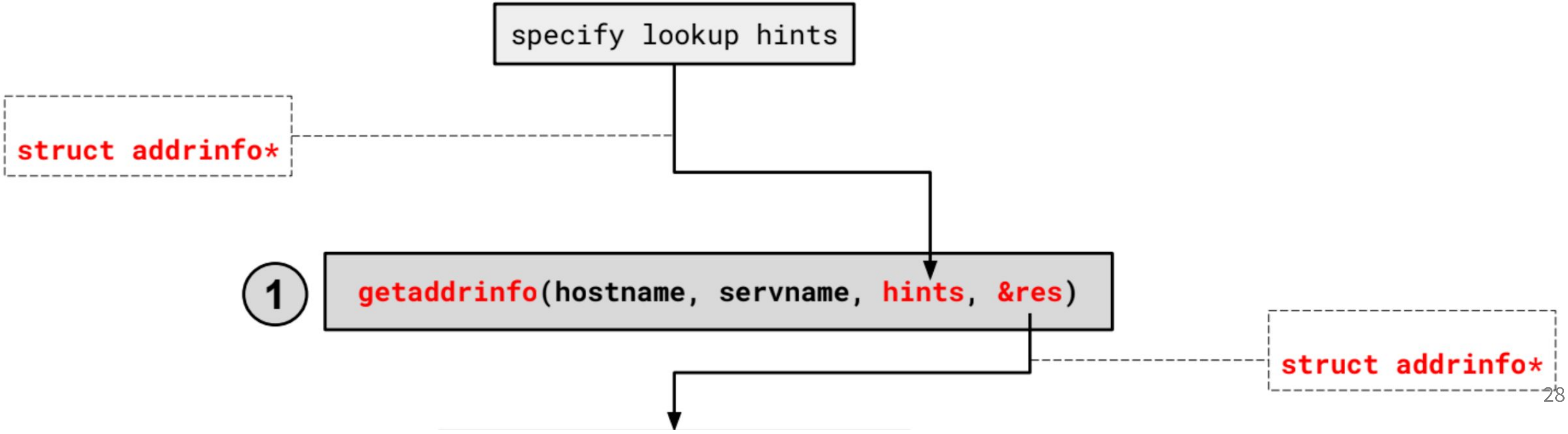
- Performs a **DNS Lookup** for a hostname



1. getaddrinfo()

```
int getaddrinfo(const char *hostname,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- Performs a **DNS Lookup** for a hostname
- Use “hints” to specify constraints (`struct addrinfo *`)
- Get back a linked list of `struct addrinfo` results



1. getaddrinfo() - Interpreting Results

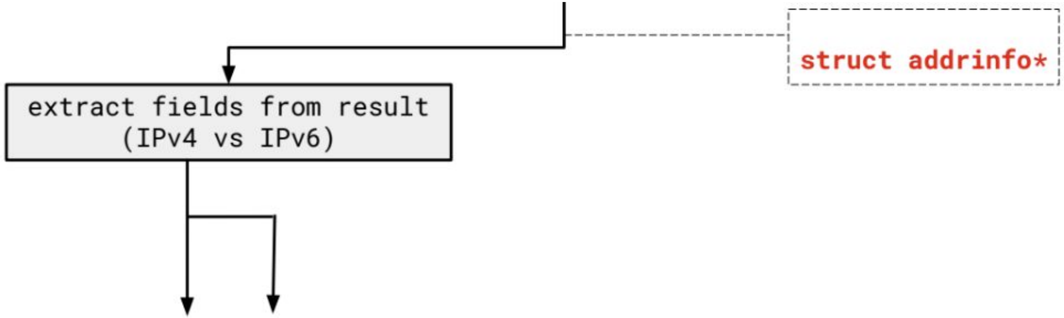
```
struct addrinfo {
    int ai_flags; // additional flags
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype; // SOCK_STREAM, SOCK_DGRAM, 0
    int ai_protocol; // IPPROTO_TCP, IPPROTO_UDP, 0
    size_t ai_addrlen; // length of socket addr in bytes
    struct sockaddr* ai_addr; // pointer to socket addr
    char* ai_canonname; // canonical name
    struct addrinfo* ai_next; // can form a linked list
};
```

- `ai_addr` points to a `struct sockaddr` describing the socket address

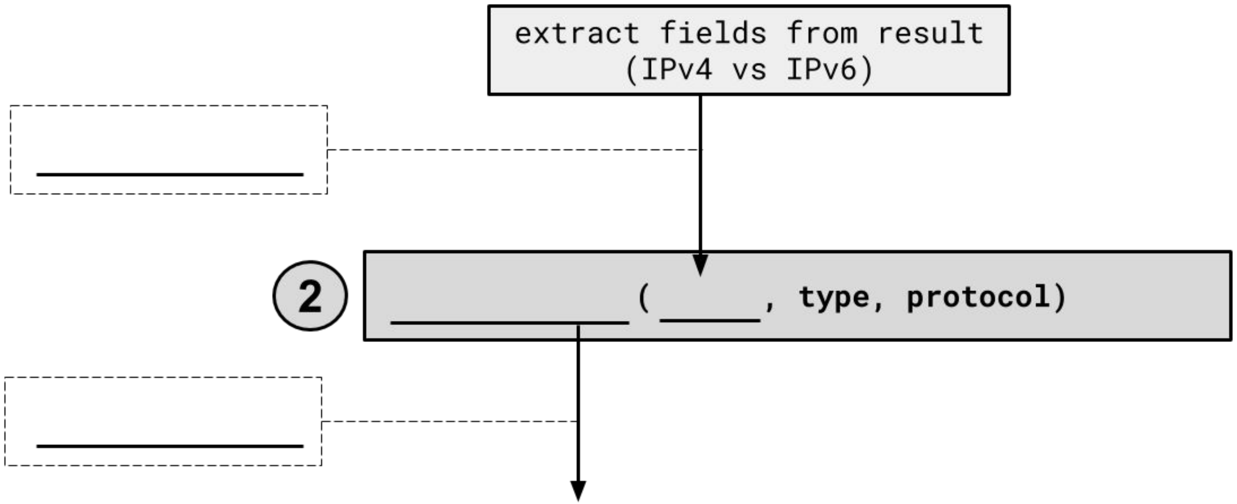
1. getaddrinfo() - Interpreting Results

With a `struct sockaddr*`:

- The field `sa_family` describes if it is IPv4 or IPv6
- Cast to `struct sockaddr_in*` `(v4)` or `struct sockaddr_in6*` `(v6)` to access/modify specific fields
- Store results in a `struct sockaddr_storage` to have a space big enough for either



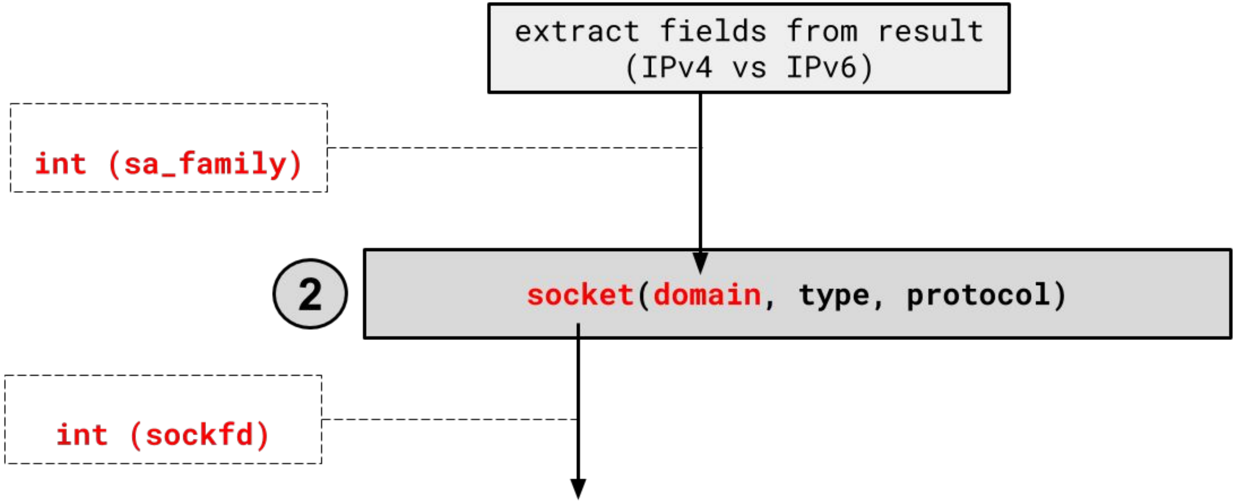
2.



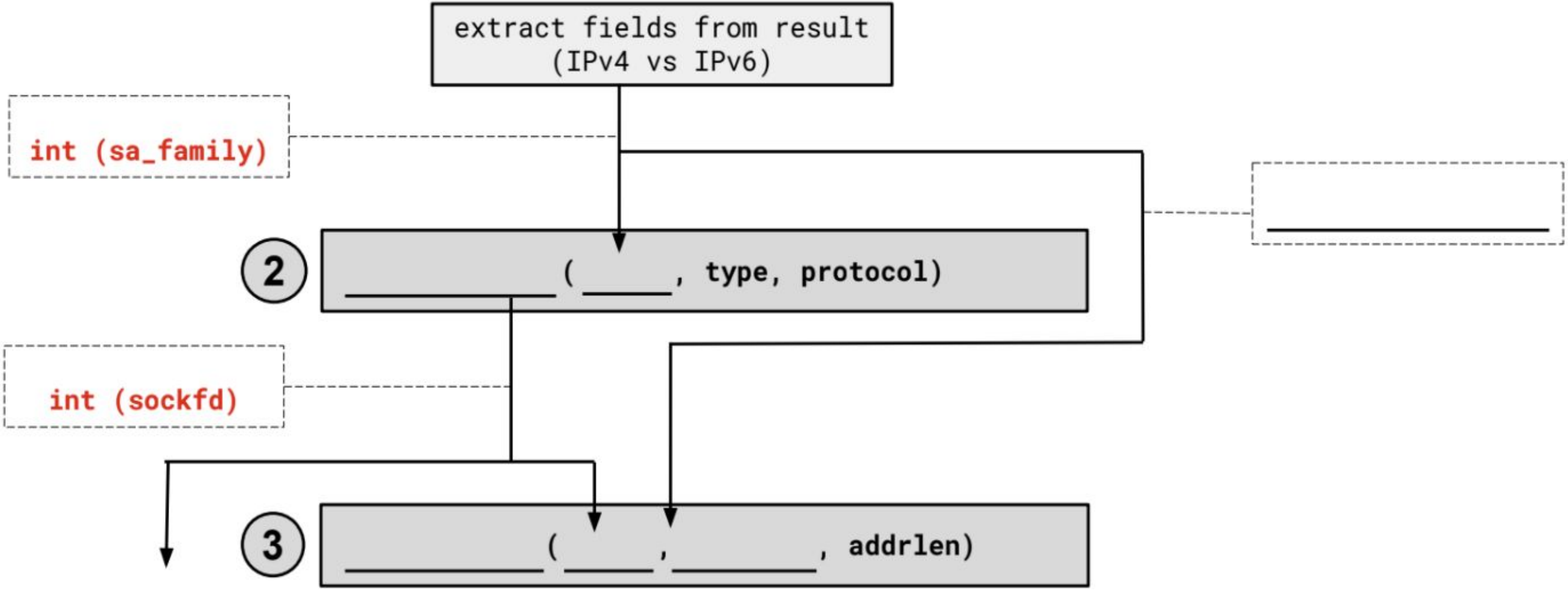
2. socket()

```
int socket(int domain,      // AF_INET, AF_INET6
           int type,        // SOCK_STREAM (TCP)
           int protocol);   // 0
```

- Creates a “raw” socket, ready to be bound
- Returns file descriptor (`sockfd`) on success, `-1` on failure



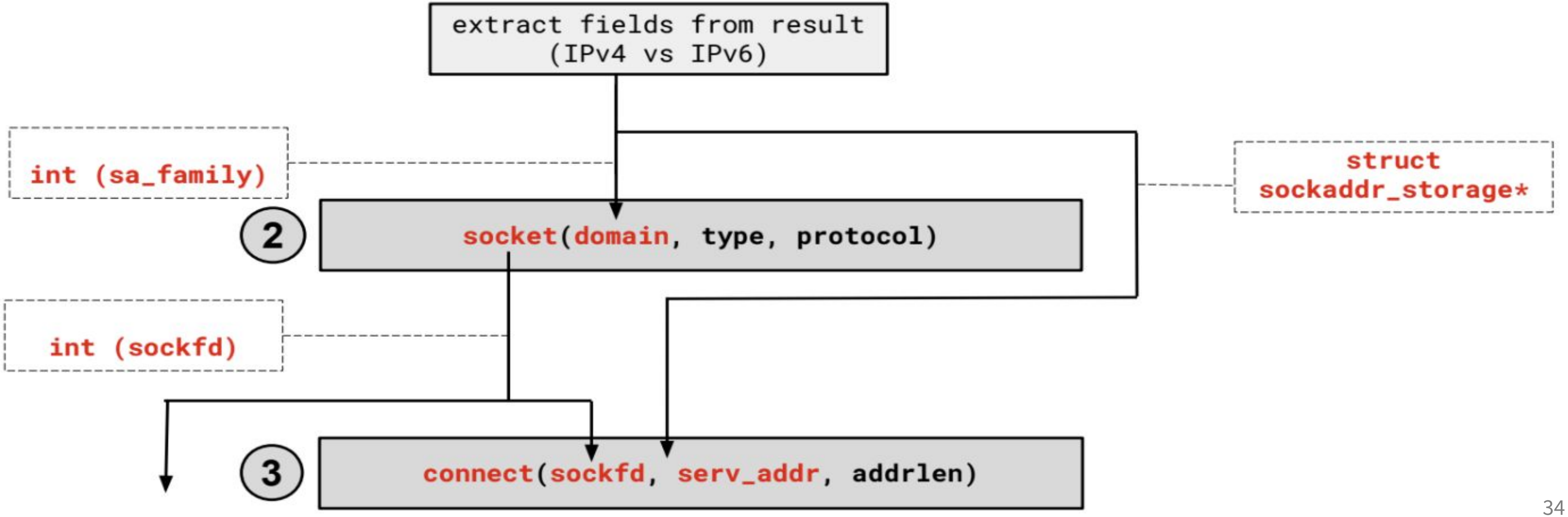
3.


extract fields from result
(IPv4 vs IPv6)
int (sa_family)
2
(, type, protocol)
int (sockfd)
3
(, , addrlen)

3. connect()

```
int connect (int sockfd,                          // from 2
             const struct sockaddr *serv_addr, // from 1
             socklen_t addrlen) ;      // size of serv_addr
```

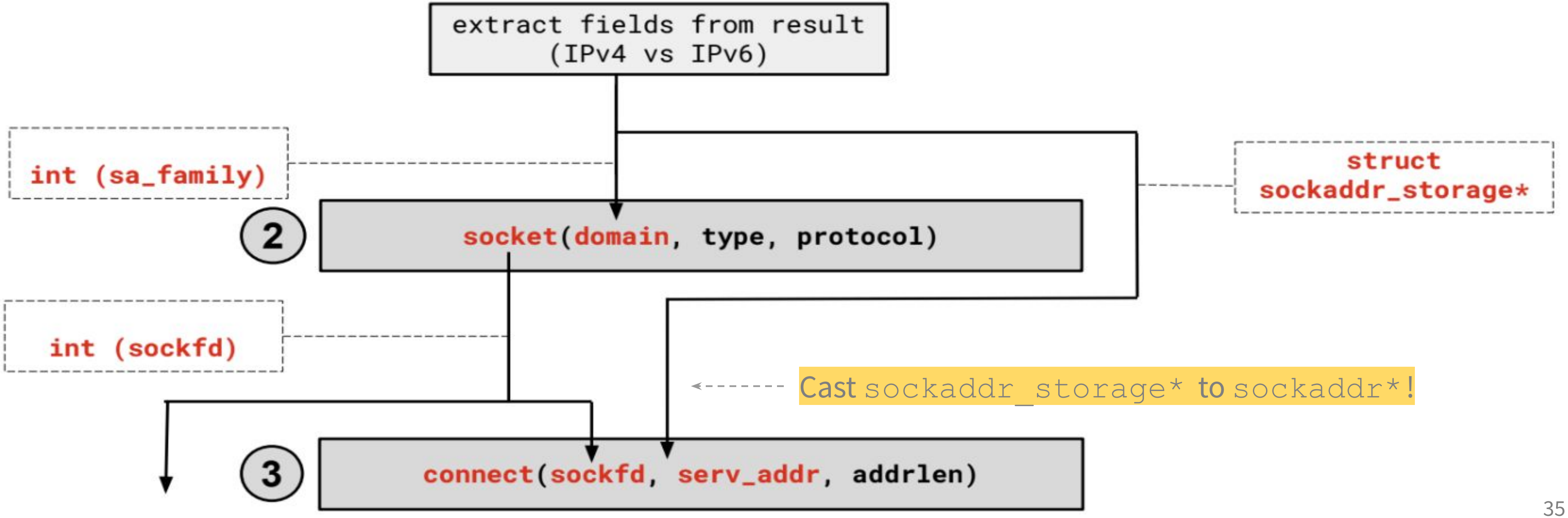
- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure



3. connect()

```
int connect (int sockfd,                         // from 2
             const struct sockaddr *serv_addr, // from 1
             socklen_t addrlen) ;      // size of serv_addr
```

- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure



4. read/write and 5. close

- Thanks to the file descriptor abstraction, use as normal!
- `read` from and `write` to a buffer, the OS will take care of sending/receiving data across the network
- Make sure to `close` the fd afterward